WHAISPER

Manual Técnico

Autores:

Cesar Arenas, Adonay Rivas

Última actualización:

23/05/2025

Índice

DOCUMENTACIÓN TÉCNICA DE WHAISPER

Versión 1.0.0.8 - Mayo 2025

Autores: César Arenas, Adonay Rivas

Resumen Ejecutivo

Whaisper es un sistema de chatbot inteligente para Telegram diseñado para facilitar la consulta y gestión de inventario comercial. Utilizando tecnologías avanzadas de inteligencia artificial como embeddings vectoriales y análisis de intención, Whaisper permite a los usuarios buscar productos, gestionar un carrito de compras y completar pedidos a través de una interfaz conversacional natural.

Características destacadas:

- Búsqueda semántica de productos mediante embeddings vectoriales
- Análisis de intención basado en IA para entender consultas en lenguaje natural
- Sistema de carrito de compras integrado en la conversación
- Panel de administración para gestión de inventario
- Soporte para mensajes de voz con transcripción automática

---

1. Información General

- **Nombre del Proyecto:** Whaisper
- **Versión del proyecto:** 1.0.1.0
- **Última actualización:** 11 de Mayo del 2025
- **Autores:** Cesar Arenas, Adonay Rivas
- **Propósito:** Proporcionar el conocimiento técnico sobre la construcción y funcionamiento del bot de Telegram para gestión de inventario y atención al cliente con capacidades de IA.

---

2. Arquitectura del Sistema

Visión general del proyecto

Whaisper es una aplicación basada en Node.js que integra tecnologías modernas para ofrecer una experiencia conversacional inteligente en Telegram. La arquitectura está diseñada con un enfoque modular que permite la separación de responsabilidades y facilita el mantenimiento y la escalabilidad.

El sistema utiliza OpenAI para tareas de comprensión del lenguaje natural y generación de respuestas, MongoDB como base de datos para almacenar productos y pedidos, y la API de Telegram para la interacción con los usuarios.

Stack Tecnológico

- **Node.js**: Entorno de ejecución para JavaScript del lado del servidor
- **Express.js**: Versión 5.1.0 - Framework web para la creación de APIs
- **OpenAI**: Versión 4.97.0 - Para integración con OpenAI API (embeddings y análisis de intención)
- **Mongoose**: Versión 8.14.1 - ODM para MongoDB
- **Node-Telegram-Bot-API**: Versión 0.66.0 - Para la integración con Telegram
- **Dotenv**: Versión 16.5.0 - Para gestión de variables de entorno
- **Multer**: Versión 1.4.5-lts.2 - Para manejo de subida de archivos
- **XLSX**: Versión 0.18.5 - Para procesamiento de archivos Excel
- **CSV-Parser**: Versión 3.2.0 - Para procesamiento de archivos CSV
- **Axios**: Versión 1.9.0 - Para peticiones HTTP
- **FS-Extra**: Versión 11.3.0 - Extensión de operaciones de sistema de archivos
- **Compute-Cosine-Similarity**: Versión 1.1.0 - Para cálculo de similitud coseno en embeddings
- **Winston**: Versión 3.10.0 - Sistema de logging avanzado

Componentes principales

Frontend

- **Telegram**: La interfaz de usuario es proporcionada por la aplicación de Telegram mediante mensajes interactivos con botones.

Backend

- **Node.js**: Servidor principal que gestiona toda la lógica de la aplicación
- **Controllers**: Conjunto de módulos que procesan tipos específicos de interacciones
- **Services**: Componentes que implementan la lógica de negocio
- **Models**: Definiciones de estructura de datos para MongoDB

Base de datos

- **MongoDB Atlas**: Almacena productos, pedidos y datos de usuarios

Servicios externos

- **OpenAI API**: Para generación de embeddings vectoriales y análisis de intención
- **Telegram Bot API**: Para la integración del bot de Telegram
- **Whisper API**: Para transcripción de mensajes de voz

Diagrama de arquitectura

```
WHAISPER/
├── /config
│   ├── constants.js        # Constantes globales
│   └── database.js         # Conexión con la base de datos
├── /controllers
│   ├── adminController.js  # Verificación del rol de usuarios
│   ├── aiController.js     # Gestión del embedding
│   ├── botController.js    # Procesamiento y respuesta de
mensajes
│   ├── cartController.js   # Gestiona el carrito de compras en
memoria
│   ├── conversationController.js # Gestión de las conversaciones
│   ├── productController.js # Selección de productos y cantidades
│   └── uploadController.js  # Manejo de subida y procesamiento de
archivos
├── /logs
│   ├── combined.log        # Todos los logs
│   └── error.log           # Registro de errores
├── /middleware
│   ├── auth.js             # Verificación de autenticación JWT
```

```
|   └── uploadMiddleware.js # Comprobaciones y configuraciones del
almacenamiento
├── /models
|   ├── user.js             # Esquema y modelo de usuarios
|   ├── articulo.js         # Esquema y modelo de los artículos
|   └── pedido.js           # Esquema y modelo de pedidos
├── /routes
|   └── api.js              # Rutas protegidas
├── /services
|   ├── botStateService.js  # Gestión del estado de la
conversación
|   ├── buttonGeneratorService.js # Botónes interactivos con el
usuario
|   ├── openaiService.js    # Integración con la API de OpenAI
|   ├── carritoService.js   # Gestiona el carrito de compras
|   ├── fileProcessingService.js # Procesamiento de archivos
|   ├── generarRespuestaComoVendedor.js # Prompts de
comportamiento
|   ├── intentAnalysisService.js # Reconocimiento de intención
usando OpenAI
|   ├── telegramServices.js # Integración de telegram
|   └── whaisperService.js  # Servicio de transcripción de audio
├── /uploads
|   └── #jsons              # Se almacena los json para los
articulos
├── /utils
|   ├── logger.js           # Mostrar logs
|   ├── helpers.js          # Calcula la similitud entre el coseno
de dos vectores
|   └── telegramUtils.js    # Manejo de errores
├── /
|   ├── generateEmbeddings.js # Genera los embeddings
|   └── index.js            # Punto de entrada
```

---

3. Entorno Técnico

Requerimientos de Hardware

Servidor de Producción Recomendado

- **CPU**: 4 núcleos (mínimo 2 núcleos)
- **RAM**: 4GB mínimo, 8GB recomendado
- **Almacenamiento**: 20GB SSD mínimo
- **Red**: Conexión de banda ancha estable (mínimo 10Mbps)

Requisitos de Escalabilidad

- Para cada 1000 usuarios activos concurrentes, considerar añadir:
    - 2 núcleos adicionales
    - 2GB de RAM adicional
    - Habilitar balanceo de carga para más de 5000 usuarios concurrentes

Entorno de Desarrollo

- **CPU**: 2 núcleos o más
- **RAM**: 4GB mínimo
- **Almacenamiento**: 10GB libre mínimo

Nota sobre Consumo de API

- **OpenAI API**: La generación de embeddings consume tokens. Considerar:
    - Modelo text-embedding-3-small: ~1500 tokens por embedding
    - Aproximadamente ~$0.0001 por artículo procesado
    - Para catálogo de 10,000 artículos: ~$1 USD costo inicial
    - Búsquedas: ~$0.00001 por consulta de usuario

Requerimientos de Software

- **Node.js**: Versión 18.0.0 o superior
- **MongoDB**: Cluster en MongoDB Atlas (driver 6.16.0)
- **NPM**: Para gestionar las dependencias
- **OpenAI API**: Cuenta con acceso a modelos de embeddings y GPT
- **Telegram Bot API**: Cuenta y token de bot creado mediante BotFather

---

4. Configuración del Sistema

Instalación

Requisitos Previos

- Node.js (v18.0.0 o superior)
- npm (v8.0.0 o superior)
- Cuenta en MongoDB Atlas (o MongoDB local v5.0+)
- Cuenta en OpenAI API con acceso a modelos de embeddings

- Bot registrado en Telegram mediante BotFather

Pasos de Instalación

Clonar el repositorio

bash

```
git clone https://github.com/usuario/whaisper.git
```

1. `cd whaisper`

2. **Instalar dependencias**
   bash
   `npm install`

Configurar variables de entorno Crear un archivo `.env` en la raíz del proyecto con el siguiente contenido:

```
# Configuración de API
PORT=3000
NODE_ENV=development

# MongoDB
MONGODB_URI=mongodb+srv://usuario:contraseña@cluster.mongodb.net/whaisper

# OpenAI
OPENAI_API_KEY=sk-xxxxxxxxxxxxxxxxxxxxxx

# Telegram
TELEGRAM_BOT_TOKEN=1234567890:ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghi
ADMIN_TELEGRAM_IDS=123456789,987654321

# JWT (para panel administrativo)
JWT_SECRET=your_super_secret_jwt_key
```

3. `JWT_EXPIRATION=24h`

4. **Generar embeddings iniciales**
   bash
   `node generateEmbeddings.js`

Iniciar el servidor

bash

```
# En desarrollo
npm run dev

# En producción
```

5. `npm start`

Despliegue

Despliegue en Servidor Linux (Ubuntu/Debian)

---

5. Base de Datos y BACKEND

Cluster

- Hicimos una kubernete con MongoDB:

- La configuración incluye una lista de IPs permitidas (whitelist) para conexión al cluster.

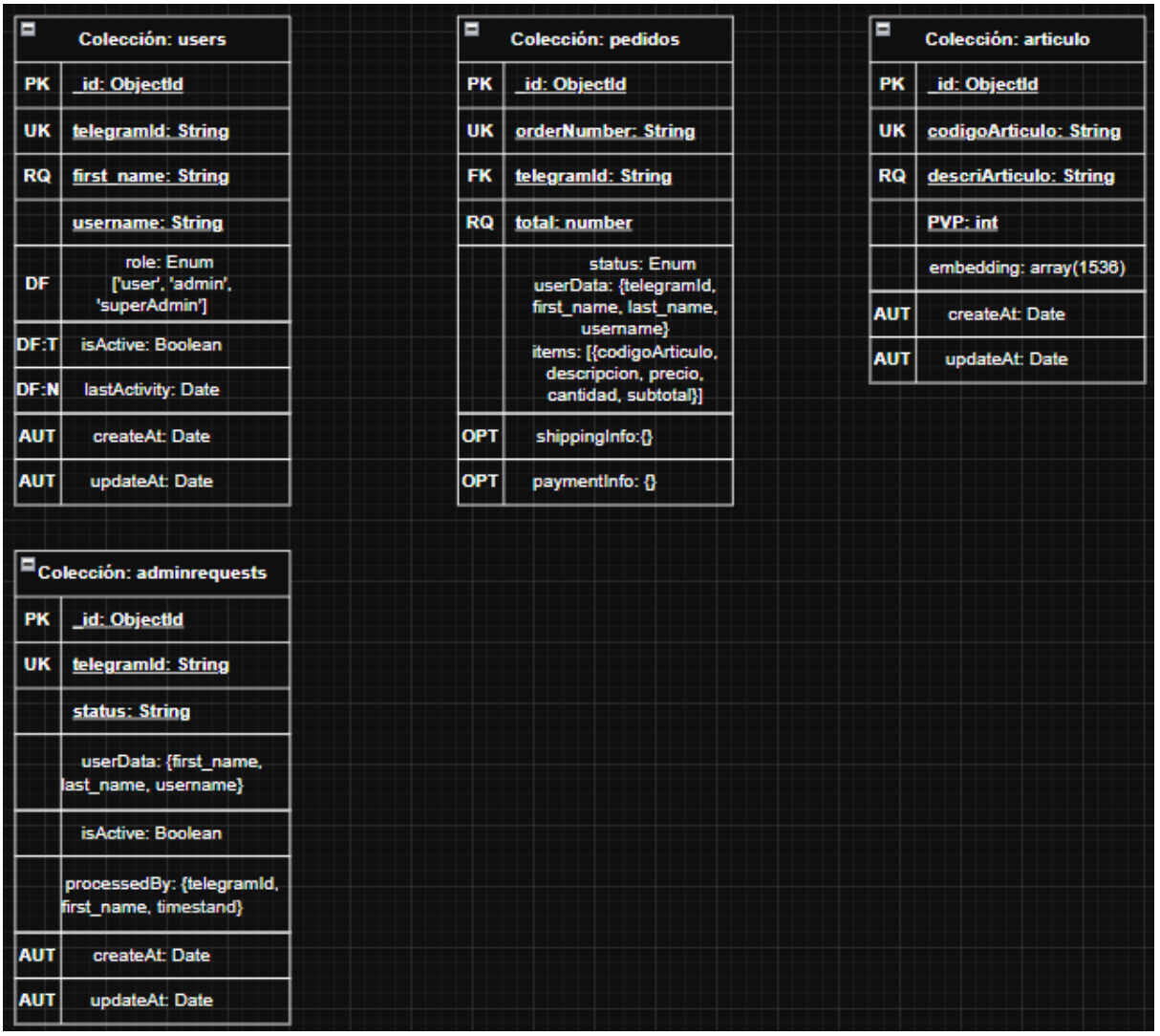
Lista de IPs Permitidas

- Para desarrollo: IPs de los desarrolladores
- Para producción: IP del servidor de despliegue
- Para CI/CD: IPs de los servicios de integración continua

Bakcend

El despliegue de la backend fue hecho en docker para mayor flexibilidad en la estructura del proyecto y menos complejidad en la configuración

Colecciones



Estrategia de Respaldo

- **Backup automático**: Configurado en MongoDB Atlas (diario)
- **Retención**: 7 días para entorno de desarrollo, 30 días para producción
- **Restauración**: Disponible a través del panel de Atlas o mediante la CLI de MongoDB

Migración y Actualización

Para migrar datos desde sistemas externos:

1. Exportar datos en formato CSV o Excel
2. Utilizar el endpoint `POST /api/upload-inventory` o la función de administrador del bot
3. Verificar la migración mediante una consulta de muestra

6. Estructura del Código

Organización del Proyecto

El proyecto sigue una estructura modular con separación clara de responsabilidades:

- **config/**: Configuraciones y constantes globales
- **controllers/**: Lógica de control para diferentes funcionalidades
- **models/**: Definiciones de esquemas para MongoDB
- **services/**: Servicios compartidos y lógica de negocio
- **middleware/**: Funciones de middleware para Express
- **routes/**: Definiciones de rutas API
- **handlers**/: Intercepta, analiza y dirige las interacciones entre el bot y los controladores.
- **temp/**: Se almacenan los audios
- **utils/**: Utilidades y funciones helper
- **uploads/**: Directorio para archivos temporales
- **logs/**: Almacenamiento de registros de la aplicación

Componentes Principales

Servicios

Los servicios implementan la lógica de negocio y funcionalidades compartidas:

- **botStateService**: Gestiona el estado de las conversaciones
- **adminService:** Servicios disponibles para el administrador
- **buttonGeneratorService**: Gestiona los botones
- **generarRespuestasComoVendedor**: Lógica de comportamiento de la IA
- **inventoryService**: Gestiona el inventario de la bd
- **orderService**: Gestiona las órdenes confirmadas
- **carritoService**: Gestiona los carritos de compra en memoria
- **openaiService**: Integración con la API de OpenAI
- **intentAnalysisService**: Análisis de intención mediante IA
- **telegramService**: Configuración del bot de Telegram
- **fileProcessingService**: Procesamiento de archivos CSV/Excel
- **whaisperService**: Transcripción de mensajes de voz
- **telegramService**: Conexión a telegram

Controladores

Los controladores manejan la interacción con los usuarios y coordinan los servicios:

- **botController**: Punto de entrada principal para mensajes de Telegram
- **adminController**: Funcionalidades de administración
- **cartController**: Gestión del carrito de compras
- **productController**: Búsqueda y selección de productos
- **conversationController**: Gestión del flujo de conversación
- **aiController**: Búsqueda semántica con embeddings
- **uploadController**: Manejo de subidas de archivos
- **audioController**: Manejo de mensajes de voz y audios
- **checkoutController**: Controlador para manejar la tramitación del pedido
- **product**/:
    - **index**: Endpoint para el manejo de todos los controladores de esta carpeta
    - **quantityModule**: Gestiona la selección del producto
    - **searchModule**: Gestiona las búsquedas de productos
    - **selectModule**: Gestiona la selección del producto
- **cart**/:
    - **addModule**: Añadir cantidades
    - **displayModule**: Gestiona como se muestra el carrito
    - **exportModule**: Genera un JSON y lo sube a la bd
    - **removeModule**: Gestiona la resta de productos en el carrito
    - **updateModule**: Gestiona las actualizaciones del carrito
    - **index**: Endpoint para el manejo de todos los controladores de esta carpeta
- **admin**/:
    - managementController: Gestión de administradores
    - **requestController**: Gestión de las peticiones para ser administrador
    - **statsController**: Gestiona las estadísticas

Modelos

Definiciones de esquemas para MongoDB:

- **articulo.js**: Productos del inventario
- **pedido.js**: Pedidos realizados
- **user.js**: Usuarios registrados
- **adminRequest**: Almacena las solicitudes para ser administrador

---

7. Módulos y Servicios

Winston: Sistema de Logging

Propósito

Winston proporciona un sistema de logging flexible y extensible que nos permite registrar información importante durante la ejecución de la aplicación, clasificarla por niveles de importancia y guardarla en diferentes formatos y destinos.

Implementación

Hemos configurado Winston con la siguiente estructura:

- **Formato de log**: [Timestamp] [NIVEL] Mensaje
- **Niveles configurados**:
    - En producción: 'info' y superiores (info, warn, error)
    - En desarrollo: 'debug' y superiores (debug, info, warn, error)

Ejemplo de configuración

javascript

```
const winston = require('winston');

const logger = winston.createLogger({
  level: process.env.NODE_ENV === 'production' ? 'info' : 'debug',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.printf(({ timestamp, level, message }) => {
      return `[${timestamp}] [${level.toUpperCase()}] ${message}`;
    })
  ),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'logs/error.log',
level: 'error' }),
    new winston.transports.File({ filename: 'logs/combined.log' })
  ]
});
```

Uso en el Proyecto

El logger se importa en los diferentes archivos como:

javascript

```
const logger = require('../utils/logger');
```

Y se utiliza en toda la aplicación:

javascript
```
logger.info('Servidor iniciado en el puerto 3000');
logger.error('Error al conectar a la base de datos: ' +
error.message);

logger.debug('Procesando mensaje: ' + mensaje);
```

Mongoose: ODM para MongoDB

Propósito

Mongoose es un Object Document Mapper (ODM) que proporciona una solución elegante basada en esquemas para modelar los datos de la aplicación. Ofrece validación, casting de tipos, hooks, consultas complejas y middleware, facilitando la interacción con MongoDB.

Implementación en Whaisper

En nuestro proyecto, Mongoose se utiliza para:

- Definir esquemas estructurados para los modelos de datos
- Establecer validaciones para la integridad de datos
- Gestionar las conexiones a MongoDB Atlas
- Implementar consultas optimizadas para búsqueda de productos

Configuración de Conexión

javascript
```
// Configuración en database.js
const mongoose = require("mongoose");
const logger = require("../utils/logger");

const mongoOptions = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  serverSelectionTimeoutMS: 15000,
  socketTimeoutMS: 45000,
  heartbeatFrequencyMS: 30000,
  retryWrites: true,
  maxPoolSize: 10,
  minPoolSize: 1,
};

async function connectWithRetry(retryAttempt = 0, maxRetries = 5)
{
  const retryDelay = Math.min(Math.pow(2, retryAttempt) * 1000,
30000);

  try {
```

```
    logger.log(`Intentando conectar a MongoDB (intento
${retryAttempt + 1}/${maxRetries + 1})...`);

    await mongoose.connect(process.env.MONGODB_URI, mongoOptions);

    logger.log("✅ MongoDB conectado correctamente");

    // Configuración de manejadores de eventos
    // ...

    return mongoose.connection;

  } catch (error) {
    // Manejo de errores y reintentos
    // ...
  }
}
```

Modelo de Artículo

javascript

```
const mongoose = require("mongoose");

const articuloSchema = new mongoose.Schema({
  CodigoArticulo: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    index: true
  },
  DescripcionArticulo: {
    type: String,
    required: true,
    trim: true
  },
  PVP: {
    type: Number,
    required: true,
    default: 0
  },
  unidades: {
    type: Number,
    default: 0,
    min: 0
  },
  embedding: {
    type: [Number],
```

```
    select: false,
    required: false
  },
}, {
  timestamps: true,
  strict: false
});

// Índices
articuloSchema.index({ CodigoArticulo: 1 }, { unique: true });
articuloSchema.index({ DescripcionArticulo: 'text' });

module.exports = mongoose.model("Articulo", articuloSchema,
"articulo");
```

Consideraciones Importantes

- **Índices Eficientes**: Implementamos índices estratégicos para mejorar el rendimiento de las consultas frecuentes.
- **Selección Condicional**: El campo `embedding` está configurado con `select: false` para excluirlo de las consultas por defecto, optimizando la transferencia de datos.
- **Validación de Datos**: Utilizamos validadores integrados de Mongoose para garantizar la integridad de los datos.
- **Conexión Resiliente**: Implementamos un sistema de reconexión automática con exponential backoff para manejar interrupciones de conexión.

OpenAI Service

Propósito

El servicio de OpenAI gestiona la interacción con la API de OpenAI para generar embeddings vectoriales de los productos y realizar el análisis de intención del usuario.

Implementación

javascript

```
const OpenAI = require("openai");

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

async function getEmbedding(text) {
  const response = await openai.embeddings.create({
    model: "text-embedding-3-small",
    input: text,
```

```
  });
  return response.data[0].embedding;
}

module.exports = { getEmbedding };
```

Modelos Utilizados

- **text-embedding-3-small**: Para generar los vectores de embedding de productos y consultas
- **gpt-4o**: Para el análisis de intención y generación de respuestas conversacionales

Consideraciones

- Los embeddings se generan una vez y se almacenan en la base de datos para optimizar costos y rendimiento
- Las consultas de análisis de intención se realizan en tiempo real para cada mensaje del usuario

Whaisper Service

Propósito

El servicio de Whaisper gestiona la transcripción de mensajes de voz y archivos de audio utilizando la API de Whisper de OpenAI, permitiendo que los usuarios interactúen con el bot mediante comandos hablados.

Características Principales

- Descarga y procesamiento de archivos de audio de Telegram
- Transcripción utilizando la API de Whisper
- Gestión de archivos temporales
- Sistema de reintentos para manejar errores de conexión

Implementación

javascript

```
async function transcribeAudio(audioFilePath, language = 'es') {
  try {
    // Verificar que el archivo existe
    if (!fs.existsSync(audioFilePath)) {
      throw new Error(`El archivo no existe: ${audioFilePath}`);
    }

    const fileStats = await fs.stat(audioFilePath);
    logger.log(`Tamaño del archivo a transcribir:
${fileStats.size} bytes`);
```

```
    if (fileStats.size === 0) {
      throw new Error('El archivo está vacío');
    }

    logger.log(`Iniciando transcripción del audio:
${audioFilePath}`);

    // Crear un stream de lectura para el archivo
    const audioFile = fs.createReadStream(audioFilePath);

    // Llamar a la API de Whisper para transcripción
    const transcription = await
openai.audio.transcriptions.create({
      file: audioFile,
      model: "whisper-1",
      language: language,
      response_format: "text"
    });

    logger.log(`Transcripción completada con éxito`);
    return transcription;
  } catch (error) {
    // Manejo de errores
    // ...
  }
}
```

Flujo de Procesamiento

1. El usuario envía un mensaje de voz a través de Telegram
2. El bot descarga el archivo de audio
3. El servicio envía el audio a la API de Whisper para transcripción
4. El texto resultante se procesa como un mensaje normal
5. Se elimina el archivo temporal

---

8. Controladores

Bot Controller

Propósito

Este es el componente principal de la aplicación, encargado de gestionar todas las interacciones del bot de Telegram. Recibe los mensajes y los dirige a los controladores específicos según su contenido y el estado actual de la conversación.

Componentes Principales

Gestión de Eventos

javascript

```
// Manejador Principal de Mensajes
bot.on("message", async (msg) => {
  // Procesamiento de mensajes
});

// Gestión de Callbacks de Botones
bot.on('callback_query', async (callbackQuery) => {
  // Procesamiento de interacciones con botones
});

// Manejo de mensajes de voz
bot.on('voice', async (msg) => {
  // Procesamiento de mensajes de voz
});
```

Flujo de Procesamiento

1. Recepción del mensaje
2. Inicialización del contexto
3. Análisis del mensaje según el estado actual
4. Delegación a controladores específicos
5. Actualización del estado de la conversación

Características Clave

- Arquitectura event-driven basada en eventos de Telegram
- Separación de responsabilidades mediante delegación a controladores especializados
- Integración con servicios de análisis de intención
- Sistema de botones interactivos para navegación
- Manejo de diferentes tipos de mensajes (texto, voz, documentos)

AI Controller

Propósito

Implementa las capacidades de búsqueda semántica mediante inteligencia artificial, utilizando vectores de embeddings para encontrar productos similares a las consultas textuales de los usuarios.

Funcionamiento Técnico

javascript
```
async function buscarArticulosSimilares(texto) {
  const inputEmbedding = await getEmbedding(texto);
  const articulos = await Articulo.find({ embedding: { $exists:
true } }).select('+embedding');

  // Filtrar artículos que no tengan embedding
  const comparaciones = articulos.map((art) => ({
    articulo: art,
    similitud: calcularSimilitud(inputEmbedding, art.embedding)
  }));

  // Ordenar por similitud y limitar a los mejores resultados
  comparaciones.sort((a, b) => b.similitud - a.similitud);
  return comparaciones.slice(0, TOP_K_RESULTS);
}
```

Características Principales

- Utiliza embeddings vectoriales para capturar relaciones semánticas
- Implementa similitud coseno para comparar vectores
- Optimiza consultas a la base de datos
- Limita resultados a los más relevantes

Cart Controller

Propósito

Gestiona todas las operaciones relacionadas con el carrito de compras, permitiendo a los usuarios visualizar, modificar y exportar su carrito.

Funciones Principales

javascript
```
// Ver el contenido del carrito
function handleCartCommand(bot, chatId) {
  // Implementación...
}

// Exportar el carrito
function handleExportCartCommand(bot, chatId) {
  // Implementación...
}

// Añadir producto al carrito
function addToCart(bot, chatId, product, quantity) {
```

```
  // Implementación...
}

// Eliminar producto del carrito
function handleRemoveFromCartCommand(bot, chatId, index) {
  // Implementación...
}
```

Características Clave

- Manejo de errores robusto
- Comunicación clara con formateo adecuado
- Botones interactivos para mejor experiencia
- Operaciones temporales para exportación

Cart/removeModule Controller

Propósito

Se ha creado para poder modularizar el código y así facilitar el mantenimiento y mejoras del carrito para este caso el eliminar.

Funciones Principales

hadleRemoveFromCartCommand: Elimina un producto
handleStartRemoveItem: Solicita el producto a eliminar
handleRemoveQuantity: Elimina la cantidad de un producto
handleStartClearCart: Registra el carrito completo
handleClearCartCommand: Limpia el carrito
handleConfirmRemove: Proceso de confirmación

Cart/addModule Controller

Propósito

Se ha creado para poder modularizar el código y así facilitar el mantenimiento y mejoras del carrito, para este caso el añadir.

Funciones Principales

addToCart: Añade el producto
handleStartAddUnits: Añade las unidades iniciales del producto
handleAddQuantity: Añade unidades extra

Cart/updateModule Controller

Propósito

Se ha creado para poder modularizar el código y así facilitar el mantenimiento y mejoras del carrito, para este caso para actualizar el modelo.

Funciones Principales

updateItemQuantity: Actualiza la cantidad de unidades
validateQuantityAgainstStock: Válida que la cantidad de unidades no exceda el stock
batchUpdatesItems: Actualiza varios ítems en el carrito a la vez

Product Controller

Propósito

Gestiona todas las operaciones relacionadas con la búsqueda, visualización y selección de productos.

Funciones Principales

javascript

```
// Búsqueda de productos
async function handleProductSearch(bot, chatId, query) {
  // Implementación...
}

// Selección de un producto
async function handleProductSelection(bot, chatId, productIndex) {
  // Implementación...
}

// Selección de cantidad
async function handleQuantitySelection(bot, chatId, productIndex,
quantity) {
  // Implementación...
}
```

Características Clave

- Búsqueda semántica con embeddings
- Respuestas naturales generadas por IA
- Interfaz de botones para selección
- Verificación de disponibilidad de stock

Product/index Controller

Propósito

Es el archivo que se usará para redireccionar a cada módulo del controlador hacia el padre.

Funciones Principales

Module.exports: Punto de acceso para la importación de los controladores

Product/quantityModule Controller

Propósito

Gestiona la selección de cantidades

Funciones Principales

generateQuantityButtonsWithStock: Genera botones con las cantidades
handleQuantitySelection: Gestiona la selección de cantidad

Product/searchModule Controller

Propósito

Gestiona las funciones relacionadas con las búsquedas del producto.

Funciones Principales

handleProductSearch: Maneja la búsqueda de productos

Product/selectModule Controller

Propósito

Gestiona las funciones relacionadas con la selección del producto.

Funciones Principales

handleProductSeleection: Gestiona la selección de un producto

Conversation Controller

Propósito

Gestiona el flujo general de la conversación, incluyendo inicios, finales, confirmaciones y cancelaciones.

Funciones Principales

javascript

```javascript
// Procesar datos del usuario
function processUserData(msg, carritoService) {
  // Implementación...
}

// Finalizar conversación
function handleEndConversation(bot, chatId) {
  // Implementación...
}

// Detectar intención de finalizar
function isEndingConversation(text) {
  // Implementación...
}
```

Características Clave

- Detección de patrones mediante expresiones regulares
- Gestión del ciclo de vida de la conversación
- Procesamiento de confirmaciones
- Limpieza de recursos temporales

Admin Controller

Propósito

Implementa funcionalidades administrativas para la gestión del inventario y otras operaciones privilegiadas.

Funciones Principales

javascript

```javascript
// Verificar permisos de administrador
function isAdmin(telegramId) {
  // Implementación...
}

// Panel de administración
function handleAdminCommand(bot, msg) {
  // Implementación...
}

// Proceso de subida de inventario
function handleUploadInventory(bot, chatId) {
  // Implementación...
}
```

Características Clave

- Control de acceso basado en IDs de Telegram
- Panel administrativo con botones interactivos
- Flujo completo para actualización de inventario
- Gestión de archivos temporales

Admin/managementController

Propósito

Gestiona las solicitudes de permisos de administrador.

Funciones Principales

showPendingRequests: Gestiona las solicitudes pendientes

Admin/requestController

Propósito

Gestiona a los administradores.

Funciones Principales

showPendingRequests: Gestiona las solicitudes pendientes
processRequest: Procesa una solicitud de administrador tanto rechazar como aprobar
handleRemoveAdmin: Gestiona la eliminación de un administrador
showAdminList: Muestra una lista de los administradores actuales

Admin/statsController

Propósito

Gestiona estadísticas de la bd.

Funciones Principales

showStatsSummary: Muestra el resumen general de las estadísticas
showPendingOrderStats: Muestra el estado de los pedidos pendientes
showCompleteOrderStats: Muestra el estado de los pedidos completados
showCanceledOrdersStat: Muestra el estado de los pedidos cancelados

showInventoryStats: Muestra de inventario

Upload Controller

Propósito

Gestiona la subida y procesamiento de archivos para el sistema de inventario, proporcionando un punto de entrada API para aplicaciones externas.

Funciones Principales

javascript

```
async function uploadFile(req, res) {
  try {
    // Verificar que hay un archivo
    if (!req.file) {
      return res.status(400).json({
        success: false,
        message: 'No se ha subido ningún archivo'
      });
    }

    const filePath = req.file.path;
    console.log(`Procesando archivo: ${filePath}`);

    // Procesar el archivo
    const articulos = await
fileProcessingService.processFile(filePath);

    // Implementación adicional...
  } catch (error) {
    // Manejo de errores...
  }
}
```

Características Técnicas

- Operación asíncrona
- Modos de vista previa y procesamiento completo
- Respuestas estructuradas
- Gestión adecuada de recursos

---

9. API e Interfaces

APIs Internas

API REST

El sistema expone algunas rutas API RESTful para integración con aplicaciones externas:

- `GET /api`: Verificación de estado de la API
- `POST /api/upload-inventory`: Endpoint para subir archivos de inventario

Interfaces Internas

A nivel de código, el sistema define varias interfaces para la comunicación entre módulos:

- **BotStateService**: Interfaz para gestionar el estado de la conversación
- **CarritoService**: Interfaz para operaciones del carrito de compras
- **FileProcessingService**: Interfaz para procesamiento de archivos

APIs Externas

El sistema integra con varias APIs externas:

Telegram Bot API

- **Propósito**: Interacción con usuarios mediante Telegram
- **Endpoint principal**: `https://api.telegram.org/bot<token>/METHOD_NAME`
- **Métodos utilizados**:
  - `sendMessage`: Enviar mensajes a usuarios
  - `sendDocument`: Enviar archivos
  - `getFile`: Obtener información de archivos subidos
  - `answerCallbackQuery`: Responder a interacciones con botones

OpenAI API

- **Propósito**: Generación de embeddings y análisis de intención
- **Endpoints utilizados**:
  - `/v1/embeddings`: Para generar embeddings vectoriales
  - `/v1/chat/completions`: Para análisis de intención y generación de respuestas
  - `/v1/audio/transcriptions`: Para transcripción de audio

MongoDB API

- **Propósito**: Almacenamiento y recuperación de datos
- **Tipo de conexión**: Mongoose ODM
- **Operaciones principales**:
    - `find`: Buscar documentos
    - `findOne`: Buscar un documento específico
    - `updateOne`: Actualizar un documento
    - `save`: Guardar un nuevo documento

---

10. Seguridad

Mecanismos de seguridad

Autenticación

- **JWT para API**: Las rutas API utilizan JSON Web Tokens para autenticación
- **ID de Telegram para Bot**: Los usuarios administradores se identifican mediante su ID de Telegram
- **Variables de Entorno**: Las credenciales se almacenan en variables de entorno, no en el código

Validación de Datos

- **Middleware de Validación**: Se validan todos los datos de entrada
- **Esquemas Mongoose**: Definen la estructura y restricciones de los datos
- **Sanitización**: Se limpian los datos de entrada para prevenir inyecciones

Protección de Recursos

- **Control de Acceso**: Verificación de permisos para acciones administrativas
- **Límite de Tamaño**: Restricción en el tamaño de archivos subidos
- **Filtro de Tipos**: Solo se permiten ciertos tipos de archivos

Políticas

Política de Acceso

- Principio de mínimo privilegio
- Acceso solo a recursos necesarios para la función
- Revisión periódica de permisos

Política de Datos

- Los datos sensibles no se almacenan en texto plano
- Se minimiza la recolección de datos personales
- Retención limitada de datos temporales

---

11. Monitorización y Mantenimiento

Logs

Sistema de Logging

- **Winston**: Utilizado para generación y gestión de logs
- **Niveles**: debug, info, warn, error
- **Formato**: [timestamp] [nivel] mensaje
- **Almacenamiento**: Consola y archivos (combined.log, error.log)

Contenido de Logs

- **Acciones de Usuario**: Todas las interacciones significativas
- **Errores**: Detalles completos con stack trace
- **Rendimiento**: Tiempos de respuesta para operaciones críticas
- **Seguridad**: Intentos de acceso no autorizados

Monitorización

Herramientas

- **MongoDB Atlas**: Dashboard para monitorizar la base de datos
- **Logs Integrados**: Análisis de logs para detección de problemas

Métricas Clave

- **Tiempo de Respuesta**: Latencia en respuestas del bot
- **Uso de Memoria**: Consumo de RAM por el proceso Node.js
- **Conexiones Activas**: Número de usuarios interactuando simultáneamente

Alertas

- **Uso Elevado de Recursos**: Alertas cuando el consumo supera umbrales
- **Indisponibilidad**: Alertas si el servicio no responde

---

12. Resolución de Problemas

Problemas comunes

Error de Conexión a MongoDB

Síntoma: Error "MongoNetworkError: failed to connect to server"
Posibles causas:

- IP no registrada en whitelist de MongoDB Atlas
- Credenciales incorrectas en variables de entorno
- Problemas de red o firewall

Solución:

1. Verificar que la IP del servidor esté en la whitelist de MongoDB Atlas
2. Comprobar las credenciales en el archivo `.env`
3. Verificar conexión de red y configuración de firewall

Error en la Generación de Embeddings

Síntoma: Error "Authentication error" al ejecutar generateEmbeddings.js
Posibles causas:

- API Key de OpenAI incorrecta o inválida
- Cuota de API agotada

Solución:

1. Verificar la API Key en el archivo `.env`
2. Comprobar el estado de la cuota en el dashboard de OpenAI
3. Ejecutar con la opción `--verbose` para obtener más detalles:
   bash
   ```
   node generateEmbeddings.js --verbose
   ```

Bot de Telegram No Responde

Síntoma: El bot no responde a mensajes
Posibles causas:

- Token de bot incorrecto
- Servidor no ejecutándose
- Webhook mal configurado (si se usa)

Solución:

1. Verificar el token en `.env`
2. Comprobar logs del servidor

Reiniciar el bot

Logs y Monitorización

Ubicación de Logs

- **Logs de aplicación:** `/logs/combined.log` y `/logs/error.log`
- **Logs de Docker:** `docker-compose logs -f whaisper`

Comandos de Diagnóstico

Verificar estado de la conexión a MongoDB:

bash

```
node -e "require('./config/database.js'); console.log('Conexión exitosa')"
```

Verificar API de OpenAI:

bash

```
node -e "const OpenAI = require('openai'); require('dotenv').config(); const openai = new OpenAI({apiKey: process.env.OPENAI_API_KEY}); async function test() { try { const resp = await openai.embeddings.create({model: 'text-embedding-3-small', input: 'test'}); console.log('API funcionando correctamente'); } catch(e) { console.error('Error:', e.message); } } test();"
```

Verificar bot de Telegram:

bash

```
node -e "const TelegramBot = require('node-telegram-bot-api'); require('dotenv').config(); const bot = new TelegramBot(process.env.TELEGRAM_BOT_TOKEN, {polling: false}); async function test() { try { const me = await bot.getMe(); console.log('Bot funcionando correctamente:', me.username); } catch(e) { console.error('Error:', e.message); } } test();"
```

---

13. Guía de Pruebas

Pruebas Manuales

Prueba de Conversación Básica

1. Iniciar chat con el bot en Telegram
2. Enviar mensaje de saludo ("Hola")
3. Verificar que el bot responde correctamente

Prueba de Búsqueda de Productos

1. Enviar consulta de producto ("Necesito un smartphone")
2. Verificar que el bot muestra resultados relevantes
3. Seleccionar un producto (botón o número)
4. Especificar cantidad
5. Confirmar adición al carrito
6. Verificar que el producto se añade correctamente

Prueba de Gestión de Carrito

1. Enviar comando `/carrito`
2. Verificar que muestra contenido actual
3. Probar modificar cantidad
4. Probar eliminar producto
5. Probar vaciar carrito
6. Verificar todas las operaciones

Prueba de Funciones de Administrador Upload

1. Enviar comando `/admin` desde cuenta autorizada
2. Verificar acceso al panel de administración
3. Probar subida de inventario con archivo de prueba
4. Verificar actualización en base de datos

Prueba de Funciones para Gestionar Administradores

1. Gestión de usuarios
2. Ver solicitudes pendientes
3. Aceptar o rechazar

Prueba de Funciones de Administrador Estadísticas

1. Resumen general
2. Pedidos pendientes
3. Pedidos completados
4. Pedidos cancelados

5. Inventario
6. Usuarios
7. Exportar datos

---

14. Anexos

Glosario de términos

- **Embedding**: Vector de números de punto flotante que representa el significado semántico de un texto.
- **Similitud Coseno**: Medida matemática que determina la similitud entre dos vectores basándose en el ángulo entre ellos.
- **TOP_K_RESULTS**: Número máximo de resultados relevantes a devolver en una búsqueda.
- **Estado de Conversación**: Etapa actual en el flujo de interacción del usuario con el bot.
- **Callback Query**: Dato enviado cuando un usuario interactúa con un botón en Telegram.
- **Whitelist**: Lista de direcciones IP autorizadas para conectarse a un servicio.

Referencias técnicas

- Documentación de Node.js
- Documentación de Mongoose
- API de Telegram Bot
- Documentación de OpenAI Embeddings

Esta documentación técnica de Whaisper proporciona una guía completa para comprender, instalar, configurar y mantener el sistema.